Chapter 1

EFFICIENT SUPPORT FOR TIME SERIES QUERIES IN DATA STREAM MANAGEMENT SYSTEMS

Yijian Bai, Chang R. Luo, Hetal Thakkar, Carlo Zaniolo
Computer Science Department
UCLA
bai,lc,hthakkar,zaniolo@CS.UCLA.EDU

Abstract There is much current interest in supporting continuous queries on data streams using generalizations of database query languages, such as SQL. The research challenges faced by this approach include (i) overcoming the expressive power limitations of database languages on data stream applications, and (ii) providing query processing and optimization techniques for the data stream execution environment that is so different from that of traditional databases. In particular, SQL must be extended to support sequence queries on time series, and to overcome the loss of expressive power due to the exclusion of blocking query operators. Furthermore, the query processing techniques of relational databases must be replaced with techniques that optimize execution of time-series queries and the utilization of main memory. The Expressive Stream Language for Time Series (ESL-TS) and its query optimization techniques solve these problems efficiently and are part of the data stream management system prototype developed at UCLA.

1. Introduction

There is much ongoing research work on data streams and continuous queries [4, 12]. The Tapestry project [6, 32] was the first to focus on the problem of 'queries that run continuously over a growing database'. Recent work in the Telegraph project [9, 21] focuses on efficient support for continuous queries and the computation of traditional SQL-2 aggregates that combine streams flowing from a network of nodes. The Tribeca system focuses on network traffic analysis [31] using operators adapted from relational algebra. The OpenCQ [19] and Niagara Systems [10] support continuous queries to monitor web sites and similar resources over the network, while the Chronicle data

model uses append-only ordered sets of tuples (chronicles) that are basically data streams [17].

The Aurora project [8] aims at building data management systems that provide integrated support for

- Data streams applications, that continuously process the most current data on the state of the environment.
- Applications on stored data (as in traditional DBs)
- Spanning applications that combine and compare incoming live data with stored data. This requires balancing real-time requirements with efficient processing of large amounts of disk-resident data.

The learning curve and complexity of writing spanning applications can be minimized if SQL is used on both data bases and data streams. This observation justifies the choice of SQL as query language made by most research projects on data streams; however, these projects often underestimate the challenges faced by SQL in this new role. For instance, the design of a general-purpose data stream language and system is the stated objective of the CQL [3] project, which introduces several SQL-based constructs with rigorous semantics [3]. Yet, CQL appears to be effective only for simple queries, and lacks the ability of supporting mining queries, sequence queries, and even some of the monotonic queries expressible in SQL which are discussed next[1].

The expressive power challenge faced by continuous query languages was elucidated in [2] where it was shown that (i) queries can be expressed by non-blocking computations iff they are monotonic, and that (ii) relational algebra (RA) and SQL are not relationally complete on data streams, since some monotonic queries[1] of relational algebra can only be expressed by RA or SQL by their blocking operators (which must be disallowed on data streams). Moreover, the seriousness of SQL problems proven by the theory are surpassed by those experience in practice, where we find that SQL cannot support many important classes of applications, including data mining and sequence queries.

The limitations of SQL with time-series queries are well-known, and have been the focus of many database research projects aiming at supporting time-series analysis and the search for interesting patterns in stored sequences [16, 23, 29, 30, 28–1]. Informix [16] was the first among commercial DBMSs to provide special libraries for time-series, that they named datablades; these libraries consist of functions that can be called in SQL queries. While other database vendors were quick to embrace it, this procedural-extension approach lacks expressive power and amenability to query optimization. To solve these problems, the SEQ and PREDATOR systems introduce a special sublanguage,

[1]These include temporal queries such as until and coalesce, and queries expressible using monotonic aggregation [2]

called SEQUIN for queries on sequences [29, 30, 28]. SEQUIN works on sequences in combination with SQL working on standard relations; query blocks from the two languages can be nested inside each other, with the help of directives for converting data between the blocks. SEQUIN's special algebra makes the optimization of sequence queries possible, but optimization between sequence queries and set queries is not supported; also its expressive power is still too limited for many application areas. To address these problems, SRQL [23] augments relational algebra with a sequential model based on sorted relations. Thus sequences are expressed in the same framework as sets, enabling more efficient optimization of queries that involve both [23]. SRQL also extends SQL with some constructs for querying sequences.

SQL/LPP is a system that adds time-series extensions to SQL [1]. SQL/LPP models time-series as attributed queues (queues augmented with attributes that are used to hold aggregate values and are updated upon modifications to the queue). Each time-series is partitioned into segments that are stored in the database. The SQL/LPP optimizer uses pattern-length analysis to prune the search space and deduce properties of composite patterns from properties of the simple patterns.

SQL-TS [26, 25] introduced simple and yet powerful extensions of SQL for finding patterns in sequences, along with techniques generalizing the Knuth-Morris-Pratt (KMP) algorithm [18] to support the optimization of such queries. The ESL-TS system, discussed next, extends those constructs to work on data streams, rather than stored data, and uses a novel implementation and optimization architecture that exploits the native extensibility of ESL and the Stream Mill system.

The paper is organized as follows. In the next section, we introduce the time-series constructs of the ESL-TS language, which is, in Section 3, compared with languages proposed in the past for similar queries. In Section 4, we discuss the native extensibility mechanisms of ESL that we use in the implementation of ESL-TS, described in Section 5. In Section 6, we provide a short overview of the query optimization techniques used in such implementation.

2. The ESL-TS Language

Our Expressive Stream Language for Time Series (ESL-TS) supports simple SQL-like constructs to specify input data stream and search for complex sequential patterns on such streams.

Suppose we have a log of the web pages clicked by a user during a session as follows:

```
STREAM Sessions(SessNo, ClickTime, PageNo, PageType) ORDER BY ClickTime;
```

Here input pages are explicitly sequenced by **ClickTime** using the ORDER BY clause. A user entering the home page of a given site starts a new session that consists of a sequence of pages clicked; for each session number, **SessNo**, the log shows the sequence of pages visited—where a page is described by its timestamp, **ClickTime**, number, **PageNo** and type **PageType** (e.g., a content page, a product description page, or a page used to purchase the item).

The ideal scenario for advertisers is when users (i) see the advertisement page for some item in a content page, (ii) jump to the product-description page with details on the item and its price, and finally (iii) click the 'purchase this item' page. This advertisers' dream pattern can be expressed by the following ESL-TS query, where 'a', 'd', and 'p', respectively, denote an ad page, an item description page, and a purchase page:

EXAMPLE 1.1 *Using the* **FROM** *clause to define patterns*

```
SELECT Y.PageNo, Z.ClickTime
FROM Sessions
        PARTITION BY SessNO AS (X, Y, Z)
WHERE X.PageType='a'
        AND Y.PageType='d'
        AND Z.PageType='p'
```

Thus, ESL-TS is basically identical to SQL, but for the following additions to the FROM clause .

- A PARTITION BY clause specifies that data for the different sessions are processed separately (i.e., as if they arrived in separate data streams.) The semantics of this construct is basically the same as the PARTITION BY construct used in SQL:1999 windows [37], which is also supported in the languages proposed by many data stream projects [4]. In this example, the PARTITION BY clause specifies that data for each **SessNO** are processed as separate streams. The pattern AS **(X, Y, Z)** specifies that, for each **SessNO**, we seek a sequence of the three tuples **X, Y, Z** (with no intervening tuple allowed) that satisfy the conditions stated in the WHERE clause.

- The AS clause, which in SQL is mostly used to assign aliases to the table names, is here used to specify a sequence of tuple variables from the specified table. By **(X, Y, Z)** we mean three tuples that immediately follow each other.

  Tuple variables from this sequence can be used in the WHERE clause to specify the conditions and in the SELECT clause to specify the output.

In the SELECT clause, we return information from both the **Y** tuple and the **Z** tuple. This information is returned immediately, as soon as the pattern is

recognized; thus it generates another stream that can be cascaded into another ESL-TS statement for processing.

Repeating Patterns and Aggregates

A key feature of ESL-TS is its ability to express recurring patterns by using a star operator. For instance, to determine the number of pages the user has visited before clicking a product description page (denoted by '**d**') we simply write:

EXAMPLE 1.2 *Number of pages visited before the product description page is clicked, provided that this count is below 20*

```
SELECT SessNo, count(*A)
FROM Sessions
        PARTITION BY SessNO
        AS (*A, B)
WHERE A.PageType <> 'd'
        AND B.PageType = 'd'
        AND count(*A) < 20
```

Thus, ***A** identifies a **maximal** sequence of clicks to pages other than 'product' pages. Then, **count(*A)** tallies up those pages and, after checking that the count is less than 20, returns **SessNo** and the associated count to the user. The maximality of the star construct is important to avoid ambiguity and the possible explosion of matches. ESL-TS supports a rich set of aggregates, as needed for time series analysis [20]; aggregates supported includes rollups, running aggregates, moving-window aggregates, online aggregates, and user-defined aggregates inherited from the AXL/ATLaS system [33]. Aggregates can only be applied to sequences defined by stars, and come in two very distinct flavors:

1 final aggregates applicable only after the star computation has completed, and

2 continuous aggregates that apply during the star computation.

For instance, **count(*A)** in Example 1.2 is a final aggregate: a sequence of pages is accepted, until a '**p**' page terminates the sequence. At that point, the condition **count(*A)** < **20** is evaluated, and if satisfied the sequence is accepted and **SessNo** and **count(*A)** for that session are returned, otherwise the sequence is rejected. Example 1.3 illustrates the use of continuous aggregates—i.e., those that return the current value of the aggregates during the computation, as per online aggregates [14]. It also illustrates how ESL-TS benefits from its ability of using standard SQL queries in combination with queries on sequences.

The previous queries were based on examples discussed in [26]. Let us now consider examples inspired by current data stream testbeds [5]. Assume we have an incoming stream **speed** sent by sensors placed on stations along the

highway, which measures the average speed of cars once every minute. Also we have a database table **stations** that has descriptions of the stations, such as "Close to Exit 111".

```
STREAM speed(stationId, speed, speedTime) ORDER BY speedTime;
TABLE stations(stationId, location);
```

A good way of determining traffic condition is to find out **jam** locations along the highway. The **jam** condition is defined as a series of decreasing speeds, which leads to a more than 70% speed reduction, from some starting speed higher than 50 mph, within a time span of at most 6 minutes (we have assumed one measurement per minute). Example 1.3 uses continuous aggregates to detect such locations. The aggregate **ccount** is the online version of count, i.e., a continuous count that returns a new value for each new input. Thus, the condition **ccount(X)** $<=$**6** is satisfied for the first 6 elements in the sequence and, upon failing on the 7th element, it brings the star sequence to completion. In general, continuous aggregates can be returned at various points during the computation of the sequence, as online aggregates do [14]; thus, they can also be used in the conditions that determine whether the current tuple must be added to the star sequence being recognized.

The two different kinds of aggregates are syntactically distinguished by the fact that, the argument of a final aggregate is prefixed by the star; while there is no star in the argument of continuous aggregates. This query also uses the aggregate **LAST**; this a built-in aggregate that always returns the final value in the star sequence (thus, in Example 1.3 it is used to return the last value of **speed** in the sequence ***Y**.)

EXAMPLE 1.3 *Find out the* **jam** *locations along the highway*

```
SELECT A.location, LAST(Y).speedTime
FROM stations AS A, speed
        PARTITION BY stationId AS (X, *Y)
WHERE X.speed > 50
        AND Y.speed < Y.previous.speed
        AND LAST(*Y).speed < 0.3*X.speed
        AND ccount(Y) <= 6
        AND X.stationId = A.stationId
```

Notice that, to retrieve the description of station locations, we use standard SQL to access database table **stations**. Also notice that we use the WHERE clause to specify conditions on both the values of attributes and those of aggregates. This is a simplification of traditional SQL (that would instead require HAVING for conditions on aggregates). This simplification is very beneficial for the users, and it has been adopted in more recent query languages such as XQuery [7].

The simplification is made possible by the lack of ambiguity associated with the sequential processing of patterns such as ***Y**. The processing is as follows: for each new tuple (i) the current values of attributes and continuous aggregates (i.e., those without the star, such as **ccount(Y)**) are evaluated and all the applicable conditions in the WHERE clause are tested, and (ii) if said conditions evaluate to true, then the computation of the star continues with the next tuple. Otherwise the evaluation of ***Y** completes and the final aggregates such as **count(*Y)** are computed and their values are used to test the applicable conditions in the where clause.

In general, therefore, we treat conditions on starred aggregates like conditions in the HAVING clause of standard SQL. Thus, for Example 1.2, the statement WHERE **count(*A)** $<$ **20** is treated like HAVING **count(A)** $<$ **20**.

Finally, the meaning of an aggregate such as **avg(*A)** would become undefined if ***A** were to contain zero or more elements, and therefore we require one or more elements in a the star construct. Therefore, ESL-TS wants to achieve both users' convenience and rigorous semantics; a formal logic-based semantics for the language constructs was presented in [24].

As a more sophisticated example, say we want to find out the course of a traffic accident from the **speed** stream. We can compute a **diff** stream from **speed** stream with the following schema[2]:

```
STREAM diff(stationId, speed_diff, speedTime) ORDER BY speedTime;
```

A tuple in **diff** specifies speed difference between cars at the current station and cars at the next station. Under normal traffic, the difference remains under a rather low value. Whenever an accident happens, we will see a sudden increase of this difference; here, we define it as a more than 2 times increase within a time span of 6 minutes. After the accident is cleared, the difference drops back to a range within 10% of the stable condition. In this query, ***Y** is the pattern when the sudden speed difference jump happens, and once the increase stops the pattern ***Z** starts to match. ***Z** matching fails when the speed difference comes back to 10% of previous difference, or 60 minutes have elapsed, at which point the pattern is returned to the user.

EXAMPLE 1.4 *Detection of traffic accidents*

```
SELECT X.stationId, FIRST(Y).speedTime,
    LAST(Z).speedTime, LAST(Z).speed_diff
    FROM diff
    PARTITION BY stationId
    AS (X, *Y, *Z)
WHERE X.speed_diff <= 15
    AND Y.speed_diff > Y.previous.speed_diff
```

[2]The computation can be easily expressed in ESL-TS

```
        AND LAST(*Y).speed_diff > 2*X.speed_diff
        AND ccount(Y) <= 6
        AND Z.speed_diff > 1.1*X.speed_diff
        AND ccount(Z) <= 60
```

Comparison with other Languages

The following example illustrates a search pattern on streams that has been previously proposed by other languages on stored sequences.

EXAMPLE 1.5 *Given a stream of events (time, name, type, magnitude) consisting of Earthquake and Volcano events, retrieve Volcano name and Earthquake name for volcano eruptions where the last earthquake (before the volcano) was greater than 7.0 in magnitude.*

```
SELECT V.name, LAST(E).name
FROM events AS (*E, V)
WHERE E.type ='Earthquake' AND V.type = 'Volcano'
        AND LAST(E).magnitude >= 7.0
```

This simple example is easily expressed in all the pattern languages proposed in the past [23, 29, 30, 28–1].

However, as illustrated in [1] most languages have problems with more complex patterns, such as the classical double-bottom queries, where given the table (name, price, time), for each stock find the W-curve (double-bottom). W-curve is a period of falling prices, followed by a period of rising prices, followed by another period of falling prices, followed by yet another period of rising prices. To make sure it is a "real" pattern we will enforce at least 5 prices in each period. In SQL/LPP+, this complex query is handled by the definition of patterns "uptrend" and "downtrend" followed by "doublebottom" as shown next.

EXAMPLE 1.6 *Double Bottom in SQL/LPP+*

```
CREATE PATTERN uptrend AS
        SEGMENT s OF quote WHICH_IS FIRST MAXIMAL, NON-OVERLAPPING
        ATTRIBUTE name AS first(s, 1).name
        ATTRIBUTE b_date AS first(s, 1).time
        ATTRIBUTE b_price AS first(s, 1).price
        ATTRIBUTE e_date AS last(s, 1).time
        ATTRIBUTE e_price AS last(s, 1).price
WHERE [ALL e IN s] (e.price >= prev(e, 1).price
        AND e.name = prev(e, 1).name)
        AND length(s) >= 5
CREATE PATTERN downtrend AS ...
/*this is similar to uptrend and omitted for lack of space*/
CREATE PATTERN double_bottom AS
        downtrend p1; uptrend p2; downtrend p3; uptrend p4 WHICH_IS ALL,
```

```
        ATTRIBUTE name IS first(p1).name
        ATTRIBUTE b_date IS first(p1).time
        ATTRIBUTE b_price IS first(p1).price
        ATTRIBUTE e_date IS last(p4).time
        ATTRIBUTE e_price IS last(p4).price
WHERE p1.name = p2.name AND p2.name = p3.name
        AND p3.name = p4.name

SELECT db.b_date, db.b_price, db.e_date, db.e_price
FROM double_bottom
```

ESL-TS can express the same pattern in much fewer lines:

EXAMPLE 1.7 *Double Bottom in ESL-TS*

```
SELECT W.name, FIRST(W).time,
        FIRST(W).price, LAST(Z).time, LAST(Z).price FROM quote
        PARTITION BY name
        SEQUENCE BY date
        AS (*W, *X, *Y, *Z)
WHERE W.price <= W.previous.price AND count(*W) >= 5
        AND X.price >= X.previous.price AND count(*X) >= 5
        AND Y.price <= Y.previous.price AND count(*Y) >= 5
        AND Z.price >= Z.previous.price AND count(*Z) >= 5
```

3. ESL and User Defined Aggregates

ESL-TS is implemented as an extension of ESL that is an SQL-based data-stream language that achieves native extensibility and Turing completeness via user-defined aggregates (UDAs) defined in SQL itself rather than in an external procedural language. In fact, using nonblocking UDAs, ESL overcomes the expressive power loss from which all data stream languages suffer because of the exclusion of blocking operators. In [2], it is shown that (i) all (and only) monotonic queries can be expressed by nonblocking computations, and (ii) using nonblocking UDAs, ESL can express all the computable monotonic functions. The practical benefits achieved by ESL's extraordinary level of theoretical power will become clear in the next section, where we will show that the pattern-searching constructs of ESL-TS can be implemented by mapping them back into the UDAs of standard ESL.

User Defined Aggregates (UDAs) are important for decision support, stream queries, and other advanced database applications [34, 4, 14]. ESL adopts from SQL-3 the idea of specifying a new UDA by an INITIALIZE, an ITERATE, and a TERMINATE computation; however, ESL lets users express these three computations by a single procedure written in SQL [33]— rather than by three procedures coded in procedural languages as prescribed by SQL-3[3].

[3] Although UDAs have been left out of SQL:1999 specifications, they were part of early SQL-3 proposals, and supported by some commercial DBMS.

Example 1.8 defines an aggregate equivalent to the standard AVG aggregate in SQL. The second line in Example 1.8 declares a local table, **state**, where the sum and count of the values processed so far are kept. Furthermore, while in this particular example, **state** contains only one tuple, it is in fact a table that can be queried and updated using SQL statements and can contain any number of tuples. Thus, INITIALIZE inserts the value taken from the input stream and sets the count to 1. The ITERATE statement updates the tuple in **state** by adding the new input value to the sum and 1 to the count. The TERMINATE statement returns the ratio between the sum and the count as the final result of the computation by the INSERT INTO RETURN statement[4]. Thus, the TERMINATE statements are processed just after all the input tuples have been exhausted.

EXAMPLE 1.8 *Defining the standard aggregate average*

```
AGGREGATE myavg(Next Int) : Real
{   TABLE state(tsum Int, cnt Int);
    INITIALIZE : {
       INSERT INTO state VALUES (Next, 1);
    }
    ITERATE : {
       UPDATE state
          SET tsum=tsum+Next, cnt=cnt+1;
    }
    TERMINATE : {
       INSERT INTO RETURN
          SELECT tsum/cnt FROM state;
    }
}
```

Observe that the SQL statements in the INITIALIZE, ITERATE, and TERMINATE blocks play the same role as the external functions in SQL-3 aggregates. But here, we have assembled the three functions under one procedure, thus supporting the declaration of their shared tables (the **state** table in this example).

deallocated just after the TERMINATE statement is completed. This approach to aggregate definition is very general. For instance, say that we want to support tumbling windows of 200 tuples [8]. Then we can write the UDA of Example 1.9, where the RETURN statements appear in ITERATE instead of TERMINATE. The UDA **tumble_avg**, so obtained, takes a stream of values as input and returns a stream of values as output (one every 200 tuples). While each execution of the RETURN statement produces here only one tuple, in general, a UDA can produce (a stream of) several tuples. Thus UDAs operate as general stream transformers. Observe that the UDA in Example 1.8 is blocking, while

[4] To conform to SQL syntax, RETURN is treated as a virtual table; however, it is not a stored table and cannot be used in any other role.

that of Example 1.9 is nonblocking. Thus, nonblocking UDAs are easily expressed in ESL, and clearly identified by the fact that *their* TERMINATE *clauses are either empty or absent*. The typical default semantics for SQL aggregates is that the data are first sorted according to the GROUP-BY attributes: thus the very first operation in the computation is a blocking operation. Instead, ESL uses a (nonblocking) hash-based implementation for the GROUP-BY calls of the UDAs. This default operational semantics leads to a stream oriented execution, whereby the input stream is pipelined through the operations specified in the INITIALIZE and ITERATE clauses: the only blocking operations (if any) are those specified in TERMINATE, and these only take place at the end of the computation.

EXAMPLE 1.9 *Average on a Tumbling Window of 200 Tuples*

```
AGGREGATE tumble_avg(Next Int) : Real
{    TABLE state(tsum Int, cnt Int);
     INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
     }
     ITERATE : {
        UPDATE state
           SET tsum=tsum+Next, cnt=cnt+1;
        INSERT INTO RETURN
           SELECT tsum/cnt FROM state
           WHERE cnt % 200 = 0;
        UPDATE state SET tsum=0, cnt=0;
           WHERE cnt % 200 = 0
     }
     TERMINATE: { }
}
```

ESL supports standard SQL, where the UDAs (defined using SQL) are called in the same way as any other built-in aggregate. As discussed above, both blocking and non-blocking UDAs can be used on database tables, however only non-blocking UDAs can be used on streams, as in the next example. For instance, given an incoming stream which contains bidding data for an online-auction web site:

```
STREAM bid(auction_id, price, bidder_id, bid_time) ORDER BY bid_time;
```

Example 1.10 continuously computes the number of unique bidders for auction with ID 1024 within a time-based sliding window of 30 minutes by applying a non-blocking UDA bidder_wcount on stream **bid** (which will be define in the next example). The first two lines in Example 1.10 illustrate stream declaration in ESL. The next two lines of Example 1.10 filter the tuples from the stream **bid** using the condition **auction_id=1024**; the tuples that survive the filter are then pipelined to the UDA **bidder_wcount**.

EXAMPLE 1.10 *UDAs and Streams in ESL*

```
STREAM bid(auction_id, price, bidder_id, bid_time)
        ORDER BY bid_time;
SELECT auction_id, bidder_wcount(bidder_id, bid_time, 30)
        FROM bid WHERE auction_id=1024;
```

In Example 1.11, we define an aggregate **bidder_wcount** that continuously returns the count of unique bidders within a sliding window of certain number of minutes, with the window size passed in as a formal parameter. Observe that the formal parameters of the UDA function are treated as constants in the SQL statements. Thus, the INSERT statement in INITIALIZE put into the table **bidders** with the constant **bidder_id** and **bid_time**. In ITERATE, we first add the bidder into the table bidders, if it is a new bidder. Then, if it is an existing bidder, we update the last seen timestamp for that bidder. Next, we delete all bidders last seen before the sliding window starts. Finally, the RETURN statement in ITERATE returns the current count of unique bidders within the sliding window.

EXAMPLE 1.11 *Continuous count of unique bidders within a sliding window of certain number of minutes*

```
AGGREGATE bidder_wcount(bidder_id, bid_time, num_min):(bcount)
{    TABLE bidders(b_id, btime);
     INITIALIZE :{
        INSERT INTO bidders VALUES(bidder_id, bid_time);
     }
     ITERATE:{
        INSERT INTO bidders VALUES(bidder_id, bid_time)
              WHERE bidder_id NOT IN (SELECT bId FROM bidders);
        UPDATE bidders SET btime = bid_time
              WHERE bidder_id = b_id;
        DELETE FROM bidders
              WHERE bid_time > (btime + num_min minutes);
        INSERT INTO RETURN
           SELECT count(b_id) FROM bidders
     }
     TERMINATE : {}
}
```

Observe that, this UDA has an empty TERMINATE, thus it is non-blocking and can be used on streams. It maintains a buffer with minimum number of tuples within the sliding window, those that are needed to ensure all the unique bidders are counted.

The power and native extensibility produced by UDAs makes them very useful in a variety of application areas, particularly those, such as data mining, that are too difficult for current O-R DBMSs [13, 22, 15, 27]. The ability of UDAs to support complex data mining algorithms was discussed in [35], where they were used in conjunction with table functions and in-memory tables to achieve performance comparable to that of procedural algorithms under the

cache mining approach. For instance in [35], a scalable decision-tree classifier was expressed in less than 20 statements. In the next section we describe how UDAs are used to implement SQL-TS.

4. ESL-TS Implementation

The ESL-TS query of Example 1.2 can be recast into the FSM of Figure 1.1, and implemented using the UDA of Example 1.12.

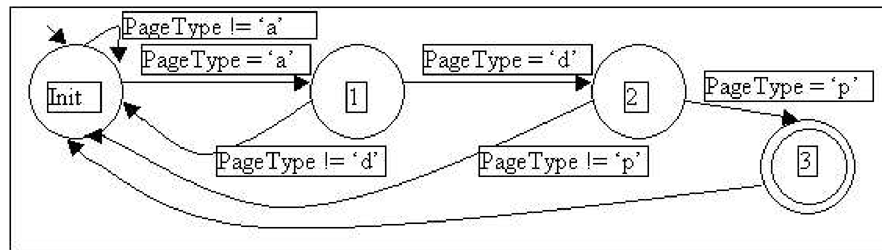

Figure 1.1. Finite State Machine for Sample Query

We can walk through the code of Example 1.12, as follows:

Lines 2 and **3:** we define local table **CurrentState** that is used to maintain the current state of the FSM, and the table **Memo** that holds the last input tuple.

Line 4: we initialize these tables as the first operation in INITIALIZE.

Line 5 and **6**: we check the first tuple to see if it is 'a'. If, and only if, this the case, the state is advanced to 1 and tuple values updated for state 1,

Line 7: we check if we have the correct input for transitioning to the next state and, in case of failure, we reset the state back to 0— this corresponds to the "Init" state in Figure 1.1. We are now in the ITERATE clause of the UDA, and this clause will be is executed for each subsequent input tuple.

Line 8: If line 7 did not execute (**sqlcode**>0, indicates the failure of the last statement), then the transition conditions hold, and we advance to the next state.

Line 9: once we transitioned into the next state (**sqlcode** =0 indicates that the last statement succeeded), we need to update the current tuple value for that state.

Line 10: if we are now in the accepting state (State 3), we simply return the tuple values.

Line 11 : once the results are returned, we must reset the FSM to its "Init" state (State 0).

EXAMPLE 1.12 *Implementation of Example 1.2*

```
1:  Aggregate find_pattern(PageNoIn int, ClickTimeIn char(16),
                 PageTypeIn char(1)): (PageNo int, ClickTime char(16))
2:  {   TABLE CurrentState(curState int);
3:      TABLE Memo(PageNo int, ClickTime char(16), State int);
      INITIALIZE: {
4:        INSERT INTO CurrentState VALUES(0);
          INSERT INTO Memo VALUES ((0, '', 1), (0, '', 2), (0, '', 3));
5:        UPDATE Memo SET PageNo = PageNoIn, ClickTime = ClickTime
              WHERE PageTypeIn = 'a' AND State = 1;
6:        UPDATE CurrentState SET curState = 1 WHERE sqlcode = 0 }
      ITERATE: {
7:        UPDATE CurrentState set curState = 0
              WHERE (curState = 0 AND pageType <> 'a')
              OR (curState = 1 AND pageType <> 'd')
              OR (curState = 2 AND pageType <> 'p');
8:        UPDATE CurrentState SET curState = curState + 1
              WHERE sqlcode > 0 AND ((curState = 0 and pageType = 'a')
              OR (curState = 1 and pageType = 'd')
              OR (curState = 2 and pageType = 'p'));
9:        UPDATE Memo SET PageNo = PageNoIn, ClickTime = ClickTimeIn
              WHERE Memo.State = (SELECT curState FROM CurrentState)
              and sqlcode =0;
10:       INSERT INTO return SELECT Y.PageNo, Z. ClickTime
              FROM CurrentState AS C, Memo AS X, Memo AS Y, Memo
              WHERE C.curState = 3 and Y.st = 2 AND Z.st = 3;
11:       UPDATE CurrentState SET curState=0 WHERE sqlcode = 0}
    }
```

Therefore it is clear how our Example 1.12 succeeds in recognizing string 'adp'. However, when our FSM input does not satisfy the patterns, we must backtrack and resume the search from previous tuples. For instance, say that our input string is 'aadp.' Then the process is as follows: we first match the first 'a', and move to State 1. But then we see another 'a' in the input string, and therefore we must restart from the Init state. Now this second 'a' brings us to State 1, and we remain in this state when we see first 'd and next 'p'. Therefore, we need the ability to resend old input tuples to the FSM machine for consideration (in the worst case we have to resend all the tuples but the oldest one). This is realized as follows:

1 We use a special UDA (the same for all ESL-TS queries), called the **buffer_manager**. This UDA passes to **find_pattern** set of tuples, as follows. The last state of the **find_pattern** UDA is checked, and if this is 0 (denoting backtracking) then teh **buffer_manager** calls **find_pattern** with the "required" old tuples. Otherwise, the **buffer_manager** calls the **find_pattern**

UDA on the current input tuple (which it also stores in **buffer** since it might needed later, after backtracking.

2 The **buffer_manager** first sends some old tuples to **find_pattern**, and then take more tuples from the input and give them to **find_pattern** with the expectation that this will resume the computation from the state in which it had left it. Thus **buffer_manager** is reentrant, and remembers the state in which it was last executed[5].

The implementation of the full ESL-TS also supports the star construct, and aggregates. The '*' construct translates to a self-loop in the FSM, and the **find_pattern** UDA was easily extended to handle such self-loops. Finally aggregates are supported, by storing an additional column in the **Memo** for each aggregate in the query (the **.previous** is implemented in a similar fashion). Optimization is discussed in the next section.

The general approach for implementing different FSMs is the same across all different FSMs, therefore we can automate this translation. The resulting UDA and ESL query can be used on both static tables and data streams. Furthermore, native SQL optimizations can be applied to both. Figure 1.1 below illustrates the corresponding FSM.

5. Optimization

The query optimization problems for continuous queries can be very different from the relational-algebra driven approach of traditional databases. Finite state automata based computation models are often used for streaming XML data [11], while the generalization of the Knuth, Morris and Pratt (KMP) text search algorithms [18] was proven very effective to mininimize execution cost of SQL-TS [26]. In ESL-TS, we are extending the KMP algorithm, to optimize memory utilization, and the execution of concurrent queries.

The KMP algorithm provides a solution of proven optimality [36] for queries such as that of Example 1.1, which searches for the sequence of three particular constant values. The algorithm minimizes execution by predicting failures and successes in the next search from those in the previous search. The algorithm takes a sequence pattern of length m, $P = p_1 \dots p_m$, and a text sequence of length n, $T = t_1 \dots t_n$, and finds all occurrences of P in T. Using an example from [18], let $abcabcacab$ be our search pattern, and $babcbabcabcaabcabcabcacabc$ be our text sequence. The algorithm starts from the left and compares successive characters until the first mismatch occurs. At each step, the ith element in the text is compared with the jth element in the pattern (i.e., t_i is compared with p_j). We keep increasing i and j until a mismatch occurs.

[5]"last state" means the value in the CurrentState table of the UDA at the end of the last call to the UDA. If the CurrentState table is outside of the find_pattern UDA then the "last state" can be retrieved from it.

j,i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t_i	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c
p_j	a	b	c	a	b	c	a	c	a	b							
				$\Uparrow$													

For the example at hand, the arrow denotes the point where the first mismatch occurs. At this point, a naive algorithm would reset j to 1 and i to 2, and restart the search by comparing p_1 to t_2, and then proceed with the next input character. But instead, the KMP algorithm avoids backtracking by using the knowledge acquired from the fact that the first three characters in the text have been successfully matched with those in the pattern. Indeed, since $p_1 \neq p_2$, $p_1 \neq p_3$, and $p_1p_2p_3 = t_1t_2t_3$, we can conclude that t_2 and t_3 can't be equal to p_1, and we can thus jump to t_4. Then, the KMP algorithm resumes by comparing p_1 with t_4; since the comparison fails, we increment i and compare t_5 with p_1:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t_i	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c
j					1	2	3	4	5	6	7	8	9	10			
p_j					a	b	c	a	b	c	a	c	a	b			
												$\Uparrow$					

Now, we have the mismatch when $j = 8$ and $i = 12$. Here we know that $p_1 \ldots p_4 = p_4 \ldots p_7$ and $p_4 \ldots p_7 = t_8 \ldots t_{11}$, $p_1 \neq p_2$, and $p_1 \neq p_3$; thus, we conclude that we can move p_j four characters to the right, and resume by comparing p_5 to t_{12}.

In general, let us assume that the search has failed on position j of the pattern and succeeded before that; it is then possible to compute at compile-time the following two functions (delivering nonnegative integers):

- $shift(j)$: this determines how far the pattern should be advanced in the input, and
- $next(j)$: this determines from which element in the pattern the checking of conditions should be resumed after the shift.

The result is of this optimization is that less backtracking is required, when $shift(j) > 0$, and fewer elements of the pattern need to be checked, when $next(j) > 0$. Thus with KMP, the complexity of searching for the first occurrence of a pattern of length m on a text of length n is is reduce to $O(m+n)$—whereas it would be $O(m \times n)$ without this optimization [18]. The Optimized Pattern Search (OPS) algorithm, proposed in [26] represents a significant generalization of the KMP algorithm in as far as we support:

- general predicates, besides the equality predicates of KMP, and

- SQL-TS patterns defined using the star and aggregates, are also fully supported.

In terms of execution speed, the OPS algorithm delivers orders of magnitude improvements over the naive search algorithm, and it therefore is being used in ESL-TS. But, in addition to this, we are using OPS to minimize memory usage. For, instance, let us return to our previous example, where we observed that the first search failed at t_4. Before that, we had succeeded at t_2, and then t_3; now, those successes are sufficient to assure that we could first discard t_2 and then t_3. Likewise, by the time we reach t_{12}, all the positions before that in memory can be discarded. This observation is of great practical value in the case of the star patterns, since a pattern ***X** can match an input stream of considerable length. Since ESL-TS only allows users to retrieve the start, the end, and aggregates functions on ***X**, we can drop from memory all the **X** values as soon as they are scanned.

A final topic of current research in ESL-TS optimization is the interaction between multiple concurrent queries. Currently, the OPS algorithm is based on the logical implications between conditions in different phases of the same query: we are now investigating how to extend our optimization to exploit implications across conditions of different queries.

6. Conclusion

Time series queries occur frequently in data stream applications, but they are not supported well by the SQL-based continuous query languages proposed by most current data stream management systems. In this paper, we have introduced ESL-TS that can express powerful time series queries by simple extensions of SQL. We have also shown that these extensions can be implemented on top of the basic ESL language—thus demonstrating the power of ESL on data stream applications and the the benefits of its native extensibility mechanisms based on UDAs. We also discussed optimization techniques for ESL-TS, and showed that they can be used to minimize execution time and memory for intra-query and inter-query optimization.

Acknowledgments

The authors wish to thank Reza Sadri for SQL-TS, and Haixun Wang for his contribution to ESL.

References

[1] *SQL/LPP: A Time Series Extension of SQL Based on Limited Patience Patterns*, volume 1677 of *Lecture Notes in Computer Science*. Springer, 1999.

[2] *Query Languages and Data Models for Database Sequences and Data Streams*, 2004.

[3] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford University, 2002.

[4] B. Babcock, S. Babu, M. Datar, R. Motawani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[5] Shivnath Babu. Stream query repository. Technical report, CS Department, Stanford University, http://www-db.stanford.edu/stream/sqr/, 2002.

[6] D. Barbara. The characterization of continuous queries. *Intl. Journal of Cooperative Information Systems*, 8(4):295–323, 1999.

[7] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu (eds.). Xquery 1.0: An xml query language–working draft 22 august 2003. Working Draft 22 August 2003, W3C, `http://www.w3.org/tr/xquery/`, 2003.

[8] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, Hong Kong, China, 2002.

[9] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

[10] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, May 2000.

[11] Yanlei Diao and Michael J. Franklin. Query processing for high-volume xml message brokering. In *VLDB 2003*, pages 261–272, 2003.

[12] Lukasz Golab and M. Tamer zsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[13] J. Han, Y. Fu, W. Wang, K. Koperski, and O. R. Zaiane. DMQL: A data mining query language for relational databases. In *Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 27–33, Montreal, Canada, June 1996.

[14] J. M. Hellerstein, P. J. Hass, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[15] T. Imielinski and A. Virmani. MSQL: a query language for database mining. *Data Mining and Knowledge Discovery*, 3:373–408, 1999.

[16] Informix. Informix: Datablade developers kid infoshelf. http://www.informix.co.za/answers/english/docs/dbdk/infoshelf, 1998.

[17] H. Jagadish, I. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *PODS*, pages 113–124, 1995.

[18] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, June 1997.

[19] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, 11(4):583–590, August 1999.

[20] G. Linoff M. J. A. Berry. *Data Mining Techniques: For Marketing, Sales, and Customer Support*. John Wiley, 1997.

[21] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–61, 2002.

[22] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *VLDB*, pages 122–133, Bombay, India, 1996.

[23] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. Beyer, and M. Krishnaprasad. Srql: Sorted relational query language, 1998.

[24] Reza Sadri. *Optimization of Sequence Queries in Database Systems*. PhD thesis, University of California, Los Angeles, 2001.

[25] Reza Sadri, Carlo Zaniolo, and Amir M. Zarkesh andJafar Adibi. A sequential pattern query language for supporting instant data minining for e-services. In *VLDB*, pages 653–656, 2001.

[26] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Optimization of sequence queries in database systems. In *PODS*, Santa Barbara, CA, May 2001.

[27] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD*, 1998.

[28] P. Seshadri. Predator: A resource for database research. *SIGMOD Record*, 27(1):16–20, 1998.

[29] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 430–441. ACM Press, 1994.

[30] Praveen Seshadri and Arun N. Swami. Generalized partial indexes. In *Proceedings of Eleventh International Conference on Data Engineering 1995*, pages 420–427. IEEE Computer Society, 1995.

[31] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, 1996.

[32] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 6 1992.

[33] Haixun Wang and Carlo Zaniolo. Using SQL to build new aggregates and extenders for object-relational systems. In *VLDB*, 2000.

[34] Haixun Wang and Carlo Zaniolo. Extending sql for decision support applications. In *Proceedings of the 4th Intl. Workshop on Design and Management of Data Warehouses (DMDW)*, pages 1–2, 2002.

[35] Haixun Wang and Carlo Zaniolo. ATLaS: A native extension of sql for data mining. In *SDM*, San Francisco, CA, 5 2003.

[36] C. A. Wright, L. Cumberland, and Y. Feng. A performance comparison between five string pattern matching algorithms. Technical Report, Dec. 1998. http://ocean.st.usm.edu/∼cawright/pattern_matching.html.

[37] Fred Zemke, Krishna Kulkarni, Andy Witkowski, and Bob Lyle. Proposal for OLAP functions. In *ISO/IEC JTC1/SC32 WG3:YGJ-nnn, ANSI NCITS H2-99-155*, 1999.